

PARA QUE EL SERVICIO PUEDA SER INVOCADO PRIMERO NECESITAMOS CONFIGURAR LA ENTRADA #100 EN LA IDT, CON DPL=3 YA QUE QUEREMOS DISPARAR LA INTERRUPCIÓN DESDE NIVEL USUARIO.

AGREGAMOS EN `idt_init()`:

`IDT_ENTRIES(100)` ✓

ESTA MACRO CONFIGURA LA IDT ENTRY CON BASE = DIRECCIÓN DEL SÍMBOLO `_IST100`. ✓

LA CONVENCION PARA PASAR LOS 3 PARÁMETROS AL SERVICIO ES MEDIANTE REGISTROS, YA QUE SE PRESERVAN AL CAMBIAR DE PRIVILEGIO.

`EAX = VIRT`, `ECX = PHY`, `EDX = TASK_SEL` ✓

EJEMPLO DE INVOCACIÓN DESDE NIVEL 3:

`main:`

`mov eax, <VIRT>`

`mov ecx, <PHY>`

`mov edx, <TASK_SEL>`

`int 100` ✓

UNA VEZ REALIZADA LA INT 100 Y ESTANDO EN NIVEL 0, CONTINÚA LA EJECUCIÓN EN EL HANDLER DE LA INTERRUPCIÓN. DESDE ACÁ SIMPLEMENTE LLAMAMOS A UNA FUNCIÓN C FOWARDEANDO LOS 3 PARÁMETROS UTILIZANDO LA CONVENCION C.

extern force\_execute

\_isr100:

push edx

push ecx

push eax

push esp ←

call force\_execute

add esp, 16

iret

Al hacer el CALL y por todo lo que pasa adentro de force\_execute el ESP ya no estaba donde yo quería (para calcular bien los offsets). Para no tener que cambiar mucho lo que había escrito, agregué a último momento pasar el ESP a la función force\_execute, con el valor que tiene justo antes del CALL (y encima debería haberlo pusheado primero porque se pushead los argumentos de derecha a izquierda). Quizás era más fácil tocar el EIP y ESP (de nivel 3) de la tarea actual directamente acá en assembler, justo antes del IRET.

LA FUNCIÓN VA A TOCAR LA PILA DE NIVEL 0 PARA QUE AL HACER iret, VAYAMOS A LA DIRECCIÓN DESEADA CON LA PILA DE NIVEL 3 RESETEADA. LE PASAMOS EL ESP (NIVEL 0) A LA FUNCIÓN C PARA PODER CALCULAR LOS OFFSETS DESDE UN ESTADO CONOCIDO,



## IMPLEMENTACIÓN DEL SERVICIO.

```
void force_execute(uint32_t virt, uint32_t phy, uint16_t task_sel,
                  uint32_t esp0) {
    tss_t* current_task_tss = &tss_tasks[current_task];
```

↓  
ESTE ARRAY CONTIENE LAS TSS DE TODAS LAS TAREAS,  
Y CURRENT\_TASK ES LA TAREA ACTIVA SEGÚN EL  
SCHEDULER. ✓

```
tss_t* other_task_tss = get_task_tss(task_sel);
```

```
uint32_t attrs = MMU_P | MMU_U | MMU_W;
```

↓  
ATRIBUTOS PARA EL MAPEO DE LAS PÁGINAS. PRESENTES, DE USER  
Y READWRITE ASUMIENDO QUE EL CÓDIGO AHÍ ESCRIBE.

SI ES CÓDIGO, NO PUEDE ESCRIBIRSE

```
mmu_map_page(current_task_tss->cr3, virt, phy, attrs);
```

```
mmu_map_page(other_task_tss->cr3, virt, phy, attrs);
```

↓  
MAPEAMOS LA PÁGINA EN EL DIRECTORY (CR3) DE CADA TAREA. EL  
ENUNCIADO NO PIDE DESMAPEARLAS. ✓

```
other_task_tss->eip = virt;
```

✓ ← ¿NO FALTA ALGO?

```
other_task_tss->esp = TASK_STACK_BASE;
```

← ¿QUÉ PILA ES ESTA?

```
other_task_tss->esp0 = TASK_KERNEL_STACK_BASE;
```

↓  
TOCAMOS DIRECTAMENTE LA TSS DE LA OTRA TAREA. CUANDO RESUMA  
SU EJECUCIÓN LO HARÁ DESDE LA DIRECCIÓN VIRT Y CON LAS PILAS  
RESETEADAS.

TASK\_STACK\_BASE = 0x08003000 DEFINIDO EN EL TALLER.

TASK\_KERNEL\_STACK\_BASE = 0x08005000 DEFINIDO POR MI.

AL INICIAR EL ESQUEMA DE PAGINACIÓN DE LAS TAREAS, SE PIDE  
UNA PÁGINA DE KERNEL PARA LA PILA NIVEL 0 Y SE MAPEA A LA  
DIR VIRTUAL TASK\_KERNEL\_STACK\_BASE - PAGE\_SIZE.

Faltó resetear los selectores de segmento de la otra tarea:

```
other_task_tss->cs = GDT_CODE_3_SEL;
```

```
other_task_tss->ds = GDT_DATA_3_SEL;
```

En vez de los defines (que no gustó mucho aunque era válido), se pueden resetear las  
pilas así (por enunciado la pila tiene 1 página disponible):

```
// Obtenemos un puntero al esp3 guardado en la pila de nivel 0.
```

```
// Para calcular el offset, recordar que en el handler del timer se hace un pushad.
```

```
uint32_t* other_task_esp3 = other_task_tss->esp0 + 44; // + 44 bytes
```

```
other_task_tss->esp = (*other_task_esp3 & 0xFFFFF000) + 0x1000;
```

```
// Para la pila de nivel 0 podemos hacerlo directamente en la TSS.
```

```
other_task_tss->esp0 = (other_task_tss->esp0 & 0xFFFFF000) + 0x1000;
```



`uint32_t * eip3 = esp0 + 16;` ✓

`uint32_t * esp3 = esp0 + 28;` ✓

`*eip3 = virt;`

`*esp3 = TASK_STACK_BASE;`

Acá también se puede sacar el define:  
`*esp3 = (*esp3 & 0xFFFFF000) + 0x1000;`

↓

TOCAMOS EN LA PILA NIVEL 0 EL EIP Y ESP DE NIVEL 3.

SS	
ESP	← ESP0 + 28
EFLAGS	
CS	
EIP	← ESP0 + 16
EDX	
ECX	
EAX	
ESP0	← ESP0

}

`tss_t * get_task_tss(uint16_t task_sel) {`

`uint32_t tss_index = task_sel >> 3;`

`uint32_t tss_addr = (gdt[tss_index].base_31_24 << 24)`

`| (gdt[tss_index].base_23_16 << 16)`

`| gdt[tss_index].base_15_0;`

`return (tss_t *) tss_addr;`

}

RETORNA UN PUNTERO A LA TSS DE LA TAREA IDENTIFICADA POR EL TASK\_SEL.

✓